In search of an API for scalable file systems: Under the table or above it? [*]

Swapnil Patil, Garth A. Gibson, Gregory R. Ganger, Julio Lopez, Milo Polte, Wittawat Tantisiroj, and Lin Xiao
Carnegie Mellon University

1 Introduction

"Big Data" is everywhere – both the IT industry and the scientific computing community are routinely handling terabytes to petabytes of data [24]. This preponderance of data has fueled the development of data-intensive scalable computing (DISC) systems that manage, process and store massive data-sets in a distributed manner. For example, Google and Yahoo have built their respective Internet services stack to distribute processing (MapReduce and Hadoop), to program computation (Sawzall and Pig) and to store the structured output data (Bigtable and HBase). Both these stacks are layered on their respective distributed file systems, GoogleFS [12] and Hadoop distributed FS [15], that are designed "from scratch" to deliver high performance primarily for their anticipated DISC workloads.

However, cluster file systems have been used by the high performance computing (HPC) community at even larger scales for more than a decade. These cluster file systems, including IBM GPFS [28], Panasas PanFS [34], PVFS [26] and Lustre [21], are required to meet the scalability demands of highly parallel I/O access patterns generated by scientific applications that execute simultaneously on tens to hundreds of thousands of nodes. Thus, given the importance of scalable storage to both the DISC and the HPC world, we take a step back and ask ourselves if we are at a point where we can distill the key commonalities of these scalable file systems.

This is not a paper about engineering yet another "right" file system or database, but rather about how do we evolve the most dominant data storage API – the file system interface – to provide the right abstraction for both DISC and HPC applications. What structures should be added to the file system to enable highly scalable and highly concurrent storage? Our goal is not to define the API calls per se, but to identify the file system abstractions that should be exposed to programmers to make their applications more powerful and portable. This paper highlights two such abstractions. First, we show how commodity large-scale file systems can support distributed data processing enabled by the Hadoop/MapReduce style of parallel programming frameworks. And second, we argue for an abstraction that supports indexing and searching based on extensible attributes, by interpreting BigTable [6] as a file system with a filtered directory scan interface.

2 Commonality between DISC & HPC

DISC and HPC are different programming environments that have developed disparate software systems. HPC is commonly characterized by distributed-memory numerical simulations using message passing groupware such as MPI and tightly-coupled low-latency memory-to-memory interconnection networks such as Infiniband. DISC is commonly characterized by web search and data analytics using parallel programming frameworks such as MapReduce and low-cost loosely-coupled commodity computers and networks such as Gigabit Ethernet. As different as these scalable computing environments may be, they also have common challenges. Both operate at a scale of thousands to tens of thousands of nodes per cluster, with an order of magnitude more CPU cores, making central the issues around parallel programming, fault tolerance, data distribution and resource load balancing.

Scalable file systems for HPC, including GPFS [28], PanFS [34], PVFS [26] and Lustre [21], and DISC, including GoogleFS [12], HDFS [15] and Amazon S3 [2], are designed to handle different workloads. But it is not clear that they need to be disjoint, and our paper makes a case that much of DISC functionality can be supported by HPC file systems.

The other direction, serving HPC applications with DISC file systems, is somewhat less clear; HPC file systems offer approximately the POSIX file system functions which many DISC file systems do not. The most often cited reason for abandoning POSIX is scalability, especially for concurrent write sharing workloads, and most HPC file systems have done at least a little relaxing of POSIX semantics as well. Perhaps by layering HPC file system middleware, designed to simplify the worst concurrent write HPC workloads, on top of an ex-

[*] In *USENIX HotCloud Workshop 2009.* June 2009, San Diego CA.

Report Documentation Page

Form Approved
OMB No. 0704-0188

Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

1. REPORT DATE **JUN 2009**	2. REPORT TYPE	3. DATES COVERED **00-00-2009 to 00-00-2009**

4. TITLE AND SUBTITLE **In search of an API for scalable file systems: Under the table or above it?**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Carnegie Mellon University,Parallel Data Laboratory,Pittsburgh,PA,15213**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR’S ACRONYM(S)
	11. SPONSOR/MONITOR’S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES
U.S. Government or Federal Rights License

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**5**	

isting DISC file system, DISC systems can serve HPC applications. For example, PLFS, a checkpointing file system used in HPC, might be stacked on a DISC file system to allow applications to perform highly concurrent write shared checkpointing [3]. Moreover, despite the author's original purposes, programmers often use Google's Chubby [4] as a file system because of its simple, strong file system semantics [1]. Perhaps DISC file systems will be drawn by their users in the direction of stronger semantics, probably at the cost of reduced scalability for the applications that need stronger semantics.

2.1 Why seek commonality?

The benefit of more commonality between HPC and DISC file systems will be programmer productivity.[1] Offering services that essentially all programmers expect, and matching their expectations for basic features and semantics, makes programming an easier task, and programs more portable and maintainable. Note that these benefits are achieved for many programmers even if implementations are not semantically equivalent. For example, programs written with no expectation of concurrent write sharing are portable across implementations with serial mutations [7], transactional changes [35], non-isolated atomic write calls [23, 28], open-close cache consistency [27] and eventually consistent caching [33], largely because programmers avoid concurrent write sharing and treat the result of it as an error no matter what the specific file system's semantics.

Of course, as a program becomes more important and emphasizes high-performance, high-availability, or high-concurrency requirements, its programmers look at the fine print of the file systems available to them for the appropriate implementation tradeoffs. Perhaps this leads to a purpose-built file system, tailored to one application, but more often it leads to an external specification, typically a namespace partition, selecting which of multiple file systems to employ. It is also common for extension APIs to be employed to adjust implementation parameters. Open modes, for example, can be used to restrict concurrent write sharing semantics.

In software engineering, this dual-interface notion is characterized as a base object and its meta-object interface [19]. We envision the scalable file system API to have this dual-interface nature. `open()`, `close()`, `read()`, `write()` would be prototypical base interfaces, and reliability/availability parameters such as replica count or RAID level would be prototypical meta interfaces. In fact, we believe that users already view 'consistency' as a meta interface, using `open()` modes or selecting among alternative file systems by pathname specification.

3 DISC data processing extensions to HPC file systems

Most DISC applications are characterized by parallel processing of massive data-sets stored in the underlying shared storage system; such distributed programming abstractions are provided by purpose-built frameworks like Mapreduce [9], Hadoop [13] and Dryad [18]. These frameworks divide a large computation into many tasks that are assigned to run on nodes that store the desired input data, and avoiding a potential bottleneck resulting from shipping around terabytes of input data. Hadoop/HDFS is an open-source implementation of Google's MapReduce/GoogleFS, and in this section we will draw examples from Hadoop's use of the HDFS cluster file system.

At a high level HDFS's architecture resembles a HPC parallel file system. HDFS stores file data and metadata on two different types of servers. All files are divided into chunks that are stored on different data servers. The file system metadata, including the per-file chunk layout, is stored on the metadata server(s). HDFS differs from HPC parallel file systems in its layout and fault tolerance schemes.

HDFS assigns chunks to compute nodes at random, while HPC file systems use a round robin layout over dedicated storage servers, and HDFS exposes a file's layout information to Hadoop. This exposed layout allows the Hadoop's job scheduler to allocate tasks to nodes in a manner that (1) co-locates compute with data where possible, and (2) load balances the work of accessing and processing data across all the nodes. Thus, the scheduler can mask sub-optimal file layout resulting from HDFS's random chunk placement policy with lots of work at each node [32]. The second big difference between HDFS and HPC file systems is its fault tolerance scheme: it uses triplication instead of RAID. With multiple copies, Hadoop scheduling has more load balancing choices. The "random" assignment of file chunks to nodes is rack-aware so the failure of an entire rack does not destroy all copies. Hadoop also uses multiple copies to re-execute (backup) computations if the original computation fails to complete in the expected time.

Given the growing importance of the Hadoop/MapReduce compute model, we ask "Could we use a commodity HPC parallel file system in-place of a custom-built DISC file system like HDFS?" While most HPC file systems use separate compute and storage systems for flexibility and manageability, most HPC parallel file systems can also be run with data servers on

[1] It is not our goal to pick a winner, but rather to strengthen the field of scalable file systems with more offerings.

		vanilla	PVFS
	HDFS	PVFS	shim
grep performance (over a 64GB data-set on 32 nodes)			
Read throughput (MB/s)	579.4	244.9	597.1
Avg CPU utilization	43%	27%	43%
Completion time (m:s)	1:45	4:08	1:46

FIGURE 1: **By exposing the file layout mapping through a non-intrusive shim layer, a commodity parallel file system (PVFS) can match the performance of HDFS for a widely used Hadoop-style workload (grep) on a 32-node setup.**

each compute node. We built a non-intrusive shim layer to plug a real-world parallel file system (the Parallel Virtual File System, PVFS [26]), into the Hadoop framework storing data on compute nodes [32]. This shim layer queries file layout information from the underlying parallel file system and exposes it to the Hadoop layer. The shim also emulates HDFS-style triplication by writing, on behalf of the client, to three data servers with every application write. Figure 1 shows that for a typical Hadoop application (grep running on 32 nodes), the performance of shim-enabled Hadoop-on-PVFS is comparable to that of Hadoop-on-HDFS. By simply exposing a file's layout information, PVFS enables the Hadoop application to run twice as fast as it would without exposing the file's layout.

Most parallel large-scale file systems, like PVFS, already expose the file layout information to client modules but do not make it available to client applications. For example, the new version 4.1 of NFS (pNFS) delegates file layout to client modules to allow the client OS to make direct access to striped files [17]. If these layout delegations were exposed to client applications to use in work scheduling decisions, as done in Hadoop/Mapreduce, HPC and pNFS file systems could be significantly more effective in DISC system usage.

4 DISC structured data extensions to HPC file systems

Once the embarrassingly parallel data processing phase is over, DISC applications often leverage structured, schema-oriented data stores for sophisticated data analysis [5, 14]. Unfortunately, traditional RDBMS designs do not provide the desired scale and performance because of the complexity of their 25-year-old code bases and because they are too heavily dependent on old technology assumptions such as the relative size of main memory and data tables, and the (mostly) OLTP dominated workloads [30]. Given this view point, scalable databases are being newly designed "from-scratch" with new combinations of database methods reflecting current technologies and application requirements [6, 29].

To scale-up, many new databases are relaxing the strong transactional guarantees – the ACID properties – by limiting atomicity to per-object or per-row mutations [6, 10], by relaxing consistency through eventual application-level inconsistency resolution or weak integrity constraints [10, 29], by providing no isolation at the system level [10], and by flexible durability guarantees for higher performance. In fact, we argue that the need for extreme scale is "clouding" the divide between file systems and databases – databases with weaker guarantees are starting to look more like file systems. In the rest of this section, we will elucidate this blur in the context of Google's BigTable [6] and show how the core BigTable properties can be incorporated into a file system using extensible attributes.

4.1 Using extensible attributes for indexing

File systems have always maintained certain attributes of each file with the file's metadata. Some file systems also offer extensible attributes: lists of name-value pairs that applications can add to a file [8, 17], although these are not yet widely expected and exploited by users. The mechanism for handling extensible attributes has been in place for some time, because access control lists are variable length, and because backup systems use extended attributes to capture implementation details enabling restored file systems to better match the properties of the original storage.

File attributes are used extensively outside the file system for various data management tasks such as backup and snapshots, regulation compliance, and general purpose search. Typically, these external systems scan the file system, one of the most expensive operation for mechanical disks, to extract attribute information and build an external index for later querying. Sometimes file systems scans are done multiple times; for example, a scan by the host OS's search tools may be followed by the user's preferred search utility, then again by the backup or snapshot tool used by the system. Moreover, the overhead of these scans may get worse with optimized data backup techniques like deduplication. It should be no surprise that file systems are under pressure to do more efficient metadata scans and full data scans, and to offer event trigger interfaces to notify a variable number of external services of changes files [11]. It should also be no surprise that consistency between the file system and the external attribute index is difficult to bound or guarantee, and that researchers have been advocating a tighter integration between file systems and attribute indices [20].

Extensible attributes are well-suited for applications that need to search based on expressions on multiple at-

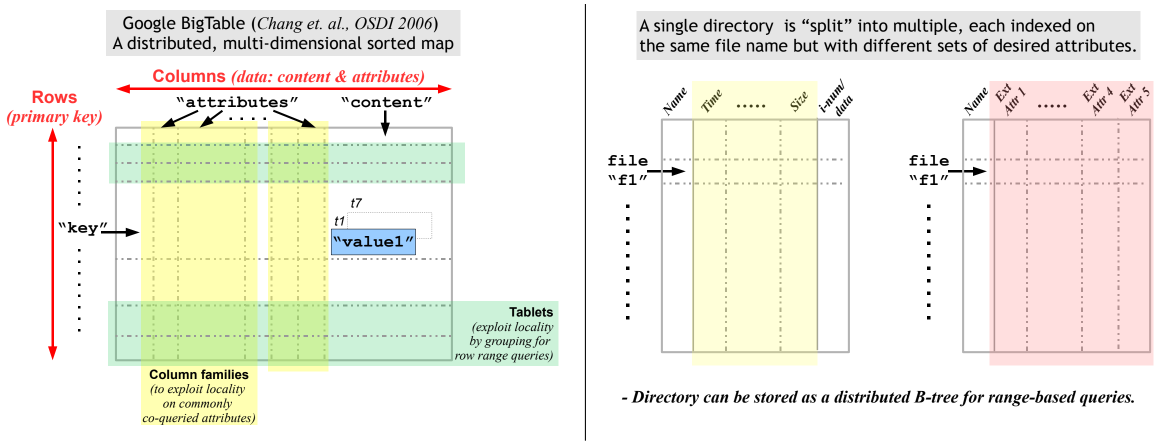
Google BigTable (Chang et. al., OSDI 2006)
A distributed, multi-dimensional sorted map
Columns (data: content & attributes)
Rows (primary key)
"attributes"
"content"
"key"
t7
t1
"value1"
Tablets (exploit locality by grouping for row range queries)
Column families (to exploit locality on commonly co-queried attributes)
A single directory is "split" into multiple, each indexed on the same file name but with different sets of desired attributes.
Name
Time
.....
Size
i-num/data
file "f1"
Name
Ext Attr 1
.....
Ext Attr 4
Ext Attr 5
file "f1"
- Directory can be stored as a distributed B-tree for range-based queries.

FIGURE 2: **BigTable-like multi-dimensional structured data stores (*left*) can be represented through file system directories (*right*) with extensible attributes that are stored redundantly with the directory index.**

tributes. Such applications can build attribute-type specific extensible indexing structures, such as KD-trees, that can optimize execution of queries about the respective attribute-type. For example, an astrophysics application can build an index based on the polar co-ordinates of the observed data (from some telescope readings) and store it in a manner that can exploit locality specific to the indexed attribute (i.e., the polar co-ordinates) for high-throughput query processing. Today such application-specific indices are opaque to the file system, stored as data in either files or embedded with the indexed data.

4.2 External attribute "tables" as file system "tables"

A leading example of an external store for extensible attributes is Google's BigTable [6], shown in the left of Figure 2. In their 'WebTable' example, each row has a web page's contents as one attribute (represented as one column) and an extensible number of other attributes (as other columns) to record all of the hyperlink anchors on other web pages that point to this web page.

BigTable can be viewed as a "middleware" database existing on top of a file system, GoogleFS, with sets of rows or with column locality groups subsets of columns of a set of rows, stored in file system files. As a database, BigTable is not relational, supports only single row transactions, with no ABORT call available to programmers, and incomplete isolation. In as much as this restricted database functionality is sufficient for attribute searching in DISC systems, it makes a case for lightweight database implementations, which we believe can be implemented in the file system itself.

Consider a file system that offers a BigTable-like query interface, as shown in the right of Figure 2. Each table could be a directory, where rows will use a file name as the primary key and columns will be the (extensible) attributes of that file stored redundantly in the directory [3, 8, 16]. Queries on a table could be implemented as extended `readdir()` interfaces taking a filter or query arguments conditional on extensible attributes stored redundantly in the directory and returning any or all columns in matching rows. Such a file system is not very different from today's file system interface, but its implementation contains a variety of open issues:

- **Indexing:** Many current file system implementations are likely to struggle with millions to billions of files in a directory, thousands to millions of attributes and high frequency of `readdir()` calls. Some file systems support large directory indices, using both hash tables [22, 28] and B-trees [31]; however, in the future, file systems may need distributed directory indices that support high mutation rates [25].

- **Scans:** File systems support full scans of directories through the `readdir()` operation that is often slow and has semantics that yield inaccurate scan results (widows and orphans). Currently most file systems store extensible attributes in blocks pointed to by their i-nodes, so each scan must perform a stat-and-seek for every file, which would not scale. By storing a redundant copy of extensible attributes in the directory, file systems can speed up attribute scans [8, 16].

- **Storage management:** Can traditional storage management be implemented efficiently as "database queries" on the directories of such a file system? How can the `find` operation for searching be accelerated? How do we effectively backup this "structured" data consisting of large number of files in file system directories?

- **Tools and optimizations:** Numerous other open challenges present themselves when dealing with massive number of files. How do we manage large numbers of small, almost empty files efficiently? How do we build mechanisms for high-performance bulk loading of directories? Can materialized views of prior scans be stored encapsulated in a file system's directory representation, and how is the space-time tradeoff managed through a file system interface?

We think it will be interesting to see how effectively a query-extended file system might satisfy BigTable-like applications in HPC and DISC, and how well it might meet the needs of storage management, regulation compliance and file system search.

5 Summary

Given the importance of the file system interface to applications of all types, this paper poses the question: *What is the "least common denominator" functionality that we should put in the file system that will allow HPC and DISC programmers to make scalable applications portable?* Based on our experience and well-known use cases, we propose a couple of new abstractions, including exposing the file layout information for the Hadoop/MapReduce applications and, most intriguingly a flexible query interface inspired by the lightweight structured data-stores used in data-intensive scalable computing today. Our goal is to start a broader discussion about the right file system functionality and interfaces for scalable computing and how the pervasive search functionality should be integrated with file systems.

Acknowledgements – The work in this paper is based on research supported in part by the Department of Energy, under Award Number DE-FC02-06ER25767, by the Army Research Office, under agreement number DAAD19-02-1-0389, by the Los Alamos National Laboratory, under contract number 54515-001-07, and by the National Science Foundation, under grants OCI-0852543, CNS-0326453 and CCF-0621499. We also thank the members and companies of the PDL Consortium (including APC, DataDomain, EMC, Facebook, Google, Hewlett-Packard, Hitachi, IBM, Intel, LSI, Microsoft, NEC, NetApp, Oracle, Seagate, Sun, Symantec, and VMware) for their interest, insights, feedback, and support.

References

[1] Private Communication with Dave Presotto, Google Inc.

[2] Amazon-S3. Amazon Simple Storage Service (Amazon S3). http://www.amazon.com/s3.

[3] Bent, J., Gibson, G., Grider, G., McClelland, B., Nowoczynski, P., Nunez, J., Polte, M., and Wingate, M. PLFS: A Checkpoint Filesystem for Parallel Applications. Tech. Rep. LA-UR 09-02117, LANL.

[4] Burrows, M. The Chubby lock service for loosely-coupled distributed systems. In *7th Symposium on Operating Systems Design and Implementation (OSDI '06)*.

[5] Cafarella, M., Chang, E., Fikes, A., Halevy, A., Hsieh, W., Lerner, A., Madhavan, J., and Muthukrishnan, S. Data Management Projects at Google. *SIGMOD Record 37*, 1 (Mar. 2008).

[6] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. Bigtable: A Distributed Storage System for Structured Data. In *7th Symposium on Operating Systems Design and Implementation (OSDI '06)*.

[7] CIFS. Common Internet File Systems. http://www.samba.org/cifs/.

[8] Custer, H. *Inside the Windows NT File System*. Microsoft Press, 1994.

[9] Dean, J., and Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. In *6th Symposium on Operating Systems Design and Implementation (OSDI '04)*.

[10] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's Highly Available Key-Value Store. In *21st ACM Symposium on Operating Systems Principles (SOSP '07)*.

[11] DMAPI. Data Storage Management (XDSM) API Specification. http://www.opengroup.org/onlinepubs/9657099/.

[12] Ghemawat, S., Gobioff, H., and Lueng, S.-T. Google File System. In *19th ACM Symposium on Operating Systems Principles (SOSP '03)*.

[13] Hadoop. Apache Hadoop Project. http://hadoop.apache.org/.

[14] Hammerbacher, J. Global Information Platforms: Evolving the Data Warehouse. PDL SDI talk at CMU on April 9, 2009 in Pittsburgh PA.

[15] HDFS. The Hadoop Distributed File System: Architecture and Design. http://hadoop.apache.org/core/docs/r0.16.4/hdfsdesign.html.

[16] HECEWG. High End Computing Extensions Working Group – man readdirplus(). http://www.opengroup.org/platform/hecewg/.

[17] IETF. NFS v4.1 specifications. http://tools.ietf.org/wg/nfsv4/.

[18] Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *2007 EuroSys Conference*.

[19] Kiczales, G. Towards a New Model of Abstraction in the Engineering of Software. In *the IMSA Workshop on Reflection and Meta-level Architectures* (1992).

[20] Leung, A., Shao, M., Bisson, T., Pasupathy, S., and Miller, E. L. Spyglass: Fast, Scalable Metadata Search for Large-Scale Storage Systems. In *FAST '09 Conference on File and Storage Technologies*.

[21] Lustre. Lustre File System. http://www.lustre.org.

[22] Mathur, A., Cao, M., Bhattacharya, S., Dilger, A., Tomas, A., and Vivier, L. The new ext4 filesystem: current status and future plans. In *Ottawa Linux Symposium 2007*.

[23] McKusick, M. K., Joy, W. N., Leffler, S. J., and Fabry, R. S. A Fast File System for UNIX. *ACM Transactions on Computer Systems 2*, 3 (Aug. 1984).

[24] Nature. Big Data. http://www.nature.com/news/specials/bigdata/.

[25] Patil, S. V., and Gibson, G. GIGA+: Scalable Directories for Shared File Systems . Tech. Rep. CMU-PDL-08-108, Carnegie Mellon University, Oct. 2008.

[26] PVFS2. Parallel Virtual File System, Version 2. http://www.pvfs2.org.

[27] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., and Lyon, B. Design and Implementation of the Sun Network Filesystem. In *Summer USENIX Conference '96*.

[28] Schmuck, F., and Haskin, R. GPFS: A Shared-Disk File System for Large Computing Clusters. In *FAST '02 Conference on File and Storage Technologies*.

[29] Stonebraker, M., Becla, J., Dewitt, D., Lim, K.-T., Maier, D., Ratzesberger, O., and Zdonik, S. Requirements for Science Data Bases and SciDB. In *4th Biennial Conference on Innovative Data Systems Research (CIDR '09)*.

[30] Stonebraker, M., and Çetintemel, U. One Size Fits All": An Idea Whose Time Has Come and Gone (Abstract). In *21st International Conference on Data Engineering (ICDE '05)*.

[31] Sweeney, A., Doucette, D., Hu, W., Anderson, C., Nishimoto, M., and Peck, G. Scalability in the XFS File System. In *USENIX Conference '96*.

[32] Tantisiroj, W., Patil, S. V., and Gibson, G. Data-intensive file systems for Internet services: A rose by any other name ... . Tech. Rep. CMU-PDL-08-114, Carnegie Mellon University, Oct. 2008.

[33] Vogels, W. Eventually Consistent. In *ACM Queue, 6(6)* (Oct. 2008).

[34] Welch, B., Unangst, M., Abbasi, Z., Gibson, G., Mueller, B., Small, J., Zelenka, J., and Zhou, B. Scalable Performance of the Panasas Parallel File System. In *FAST '08 Conference on File and Storage Technologies*.

[35] WinFS. Microsoft WinFS. http://www.microsoft.com.